Journal of Universal Computer Science, vol. 13, no. 12 (2007), 1805-1834
submitted: 15/4/07, accepted: 31/7/07, appeared: 1/12/07 © J.UCS

ODEDialect: a Set of Declarative Languages for Implementing Ontology Translation Systems

Oscar Corcho
(Ontology Engineering Group, Universidad Politécnica de Madrid, Spain
ocorcho@fi.upm.es)

Asunción Gómez-Pérez
(Ontology Engineering Group, Universidad Politécnica de Madrid, Spain
asun@fi.upm.es)

Abstract: Implementing ontology translation systems is a complex task that requires taking many types of translation decisions, which are usually hidden inside their source code. In order to ease building, maintaining and understanding ontology translation systems, we propose ODEDialect, a set of languages to express translation decisions declaratively and at different layers: lexical, syntax, semantics, and pragmatics. This paper describes the three languages that comprise ODEDialect: ODELex, which allows expressing transformations in the lexical layer; ODESyntax, which allows expressing transformations in the syntax layer; and ODESem, which allows expressing transformations in the semantic and pragmatic layers.

Keywords: ODEDialect, Ontology Language, Translation
Categories: I.2.4, M.2, M.8

1 Introduction

An ontology is defined as a "formal explicit specification of a shared conceptualisation" [Gruber, 1993], that is, an ontology must be machine readable (it is formal), all its components must be described clearly (it is explicit), it describes an abstract model of a domain (it is a conceptualisation) and it is the product of a consensus (it is shared).

Ontologies can be implemented in varied ontology languages, which are usually divided in two groups: classical and ontology markup languages. Among the classical languages used for ontology construction we can cite (in alphabetical order): CycL [Lenat and Guha, 1990], FLogic [Kifer et al, 1995], KIF [Genesereth and Fikes, 1992], LOOM [MacGregor, 1991], OCML [Motta, 1999], and Ontolingua [Gruber, 1992]. Among the ontology markup languages, used in the context of the Semantic Web, we can cite: RDF [Lassila and Swick, 1999], RDF Schema [Brickley and Guha, 2004], and OWL [Dean and Schreiber, 2004]. Each of these languages has its own syntax, its own expressiveness, and its own reasoning capabilities, provided by different inference engines. Languages are also based on different knowledge representation paradigms and combinations of them (frames, first order logic, description logic, semantic networks, topic maps, conceptual graphs, etc.).

A similar situation applies to ontology tools: several ontology editors and ontology management systems can be used to develop ontologies. Among them we can cite (in alphabetical order): KAON [Maedche et al., 2003], OilEd [Bechhofer et al., 2001], OntoEdit [Sure et al., 2002], the Ontolingua Server [Farquhar et al., 1997], OntoSaurus

[Swartout et al., 1997], Protégé [Noy et al., 2000], WebODE [Arpírez et al., 2003], and WebOnto [Domingue, 1998]. As in the case of languages, the knowledge models underlying these tools have their own expressiveness and reasoning capabilities, since they are also based on different knowledge representation paradigms and combinations of them. Besides, ontology tools usually export ontologies to one or several ontology languages and import ontologies coded in different ontology languages.

There are important connections and implications between the knowledge modelling components used to build an ontology in such languages and tools, and the knowledge representation paradigms used to represent formally such components. With frames and first order logic, the knowledge components commonly used to build ontologies are [Gruber, 1993]: classes, relations, functions, formal axioms, and instances; with description logics, they are usually [Baader et al., 2003]: concepts, roles, and individuals; with semantic networks, they are: nodes and arcs between nodes; etc.

The **ontology translation problem** [Gruber, 1993] appears when we decide to reuse an ontology (or part of an ontology) with a tool or language that is different from those ones where the ontology is available. If we force each ontology-based system developer, individually, to commit to the task of translating and incorporating to their systems the ontologies that they need, they will require both a lot of effort and a lot of time to achieve their objectives [Swartout et al., 1997]. Therefore, ontology reuse in different contexts will be highly boosted as long as we provide ontology translation services among those languages and/or tools. This is important to improve ontology reuse in different contexts, as described in works like [Tempich et al., 2005; Valente et al., 1999, Fernández-López et al., 2000], among others, where ontology reuse and reengineering are identified as important parts of the ontology development process.

Several ontology translation systems can be found associated to the most relevant ontology editors and platforms, like Protégé, SWOOP, the NeOn toolkit, KAON, WebODE, etc. They are mainly aimed at importing ontologies implemented in a specific ontology language to an ontology tool, or at exporting ontologies modelled with an ontology tool to an ontology language, so as to give support to the ontology reuse and reengineering, and implementation activities of the aforementioned approaches.

A smaller number of ontology translation systems are aimed at transforming ontologies between ontology languages or between ontology tools, giving support as well to the aforementioned ontology reuse and reengineering activities. The most well-known translation systems in this category are those available in the Ontolingua server to translate from and to KIF (and Ontolingua). Others are also available for transforming DAML+OIL into OWL, OWL into RDF(S), OCML into OWL and viceversa, etc.

Since ontology tools and languages have different expressiveness and reasoning capabilities, **translations between them are not straightforward or easily reusable**. They normally require taking many **decisions at different levels**, which range from low layers (i.e., how to transform a concept name identifier from one format to another) to higher layers (i.e., how to transform a ternary relation among concepts to a format that only allows representing binary relations between concepts).

Existing approaches to build ontology translation systems, which are described in the related work section, do not take into account such a layered structure of translation decisions. Besides, in these systems **translation decisions are usually hidden inside their programming code**. Both aspects make it difficult to understand how ontology translation systems work.

To ameliorate this problem, in this paper we propose ODEDialect, a set of languages that allow expressing declaratively translation decisions at different levels: lexical, syntax, semantics, and pragmatics (this structure of layers is based on the theory of signs [Morris, 1938]). This paper describes the three languages that comprise ODEDialect: ODELex, which allows expressing transformations in the lexical layer; ODESyntax, which allows expressing transformations in the syntax layer; and ODESem, which allows expressing transformations in the semantic and pragmatic layers.

This paper is structured as follows: section 2 describes the four layers where ontology translation problems may appear, with examples of how transformations have to be made at each layer. Section 3 describes the three languages that comprise ODEDialect, with their grammars and examples of their use. Section 4 presents the main conclusions of our work and future work. Section 5 presents related work on ontology translation.

2 Ontology translation layers

The ODEDialect language allows expressing translation decisions in four different layers, which are based on existing work on formal languages and the theory of signs [Morris, 1938]. Such works consider the existence of several levels in the definition of a language: syntax (related to how the language symbols are structured), semantics (related to the meaning of those structured symbols), and pragmatics (related to the intended meaning of the symbols, that is, how symbols are interpreted or used).

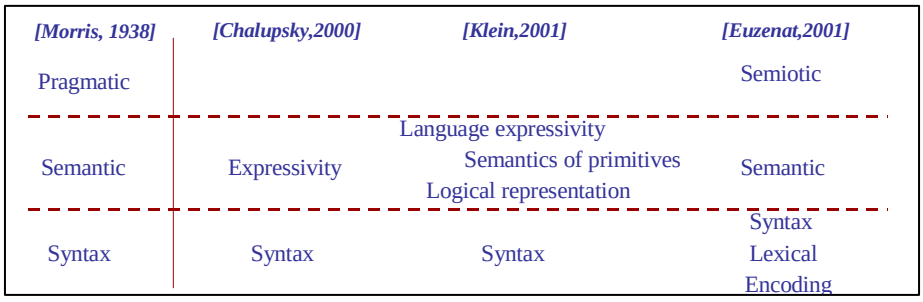

Figure 1: Relationships between classifications of semantic interoperability problems.

In the context of semantic interoperability, some authors have proposed classifications of the problems to be faced when managing different ontologies in, possibly, different formats. We will enumerate only the ones that are due to

differences between the source and target formats[1]. [Euzenat, 2001] distinguishes the following non-strict levels of language interoperability: encoding, lexical, syntactic, semantic, and semiotic. [Chalupsky, 2000] distinguishes two layers: syntax and expressivity (aka semantics). [Klein, 2001] distinguishes four levels: syntax, logical representation, semantics of primitives, and language expressivity, where the last three levels correspond to the semantic layer identified in the other classifications. Figure 1 shows the relationship between these layers.

The layers described in this section are mainly based on Euzenat's classification. This classification is the only one in the context of semantic interoperability that deals with pragmatics (although Euzenat uses the term "semiotics" to refer to it). However, we consider that it is not necessary to split the lexical and encoding layers when dealing with ontologies, and consider them as a unique layer, called "lexical".

In the next sections we describe the types of translation problems that can be usually found in each of these layers and will show some examples of common transformations performed in each of them.

2.1 Lexical layer

The lexical layer deals with the "ability to segment the representation in characters and words (or symbols)" [Euzenat, 2001]. Different languages and tools normally use different character sets and grammars to generate their terminal symbols. Therefore, in this layer we deal with transformations of ontology component identifiers, of pieces of text used for natural language documentation purposes, and of values.

Lexical transformations mainly consist in replacing non-allowed characters by other allowed ones (e.g., class identifiers in Protégé can contain blank spaces – for instance, *Travel Agency* –, while this is not possible in Ontolingua – it would be transformed to *Travel-Agency* –), or replacing identifiers that are reserved keywords in a format to other ones that are not reserved keywords (e.g., if an OWL ontology contains the class *:THING*, it cannot be transformed to Protégé).

Other sources of problems in lexical transformations are related to the scope of the ontology component identifiers in the source and target formats, and the restrictions related to overlapping identifiers. These problems appear when, in the source format, a component is defined inside the scope of another component, and hence its identifier is local to the latter, and in the target format the correspondent component has a global scope. As a consequence, there could be clashes of identifiers in case that in the source format two components have the same identifier.

2.2 Syntax problems

This layer deals with the "ability to structure the representation in structured sentences, formulas or assertions" [Euzenat, 2001]. Ontology components in each language or tool are defined with different grammars. Hence, this translation layer deals with the problems related to how symbols are structured in the source and target formats, taking into account their derivation rules for ontology components.

---

[1] Semantic interoperability problems do not only appear because ontologies are available in different formats, but also because of their content, their ontological commitments, etc. We only focus on problems related exclusively to differences among ontology languages or tools.

The following types of transformations are included in this layer: transformations of ontology component definitions according to the grammars of the source and target formats (e.g., the grammar to define a concept in Ontolingua is different than that of OCML) and transformations of datatypes (e.g., the datatype "date" in WebODE must be transformed to the datatype "&xsd;date" in OWL).

With regard to the syntax differences between formats, we can distinguish basically three groups of languages and tools: Lisp-based formats (usual in many classical languages and tools, such as Ontolingua, LOOM, or OCML), XML-based formats (usual in ontology markup languages), and ad-hoc text formats (like in FLogic). Besides, several languages and tools provide ontology management APIs in different programming languages, such as Java, C++, Lisp, etc., which could be considered as another form of syntax for representing ontologies.

With regard to datatypes, we can distinguish basically two groups of languages and tools: those with their own datatypes (integer, float, number, string, etc.), and those that allow using XML Schema datatypes (normally ontology markup languages).

2.3 Semantic problems

The semantic layer deals with the "ability to construct the propositional meaning of the representation" [Euzenat, 2001]. Different ontology languages and tools can be based on the same KR paradigm, on different KR paradigms (frames, semantic networks, first order logic, conceptual graphs, etc.) or on combinations of them.

In this layer we deal not only with simple transformations (e.g., FLogic concepts are transformed into Ontolingua and OWL classes), but also with complex transformations of expressions that are usually related to the fact that the source and target formats are based on different KR paradigms (e.g., WebODE disjoint decompositions are transformed into subclass-of relationships and PAL[2] constraints in Protégé, FLogic instance attributes attached to a concept are transformed into datatype properties in OWL and unnamed property restrictions for the class).

Most of the work on ontology translation done so far has been devoted to solving the problems that arise in this layer. For example, in the literature we can find several formal, semi-formal, and informal methods for comparing ontology languages and ontology tools' knowledge models ([Baader, 1996], [Borgida, 1996], [Euzenat and Stuckenschmidt, 2003], [Corcho and Gómez-Pérez, 2000], [Knublauch, 2003], etc.), which aim at helping to decide whether two formats have the same expressiveness or not, so that knowledge can be preserved in the transformation. And some of these approaches can be also used to decide whether the reasoning mechanisms present in both formats will allow inferring the same knowledge in the target format.

Basically, these studies allow analysing the expressiveness (and, in some cases, the reasoning mechanisms) of the source and target formats, so that we can know which types of components can be translated directly from a format to another, which types of components can be expressed using other types of components from the target format, which types of components cannot be expressed in the target format, and which types of components can be expressed, although losing part of the knowledge represented in the source format.

---

[2] Protégé Axiom Language

In summary, the catalogue of problems found in this layer are mainly related to the different KR formalisms in which the source and target formats are based. This does not mean that translating between two formats based on the same KR formalism is straightforward, since there might be differences in the types of ontology components that can be represented in each of them. This is specially important in the case of DL languages, since many different combinations of primitives can be used in each language, and hence many possibilities exist in the transformations between them, as shown in [Euzenat and Stuckenschmidt, 2003]. However, the most interesting results appear when the source and target KR formalisms are different.

2.4 Pragmatic problems

This layer deals with the "ability to construct the pragmatic meaning of the representation (its meaning in context)". In this layer we deal with transformations to be made in the resulting ontology so that human users and ontology-based applications will notice as less differences as possible with respect to the ontology in the original format, either in one-direction transformations or in cyclic transformations.

Transformations in this layer require, among other, the following: adding special labels to ontology components so as to preserve their original identifier in the source format (e.g., adding an own slot to all the Protégé classes obtained when transforming an ontology from another tool or language); transforming sets of expressions into more legible syntactic constructs in the target format (e.g., transforming a set of Protégé PAL constraints into a single class description); somehow "hiding" completely or partially some ontology components that were not defined in the source ontology, but which have been created as part of the transformations (such as the anonymous classes that are usually created when transforming between a DL-based format to a frame-based format); etc.

2.5 Relationships between ontology translation layers

Figure 2 shows an example of a transformation from the ontology platform WebODE to the language OWL DL. In this example, we have to transform two ad hoc relations with the same name (*usesTransportMean*) and with different domains and ranges (a *flight* uses an *airTransportMean* and a *cityBus* uses a *bus*). In OWL DL the scope of an object property is global to the ontology, and so we cannot define two different object properties with the same name. The example shows that translation decisions have to be taken at all layers, and that the decisions taken at one layer can influence on the decisions to be taken at the others, hence showing the complexity of this task.

Option 1 is driven by semantics: to preserve semantics in the transformation, two different object properties, with different identifiers, are defined. Option 2 is driven by pragmatics: only one object property is defined from both ad hoc relations, since we assume that they refer to the same meaning, but some knowledge is lost in the transformation (the one related to the object property domain and range). Finally, option 3 is also driven by pragmatics, with more care on the semantics: again, only one object property is defined, and its domain and range is more restricted than in option 2, although we still lose the exact correspondence between each domain and range.

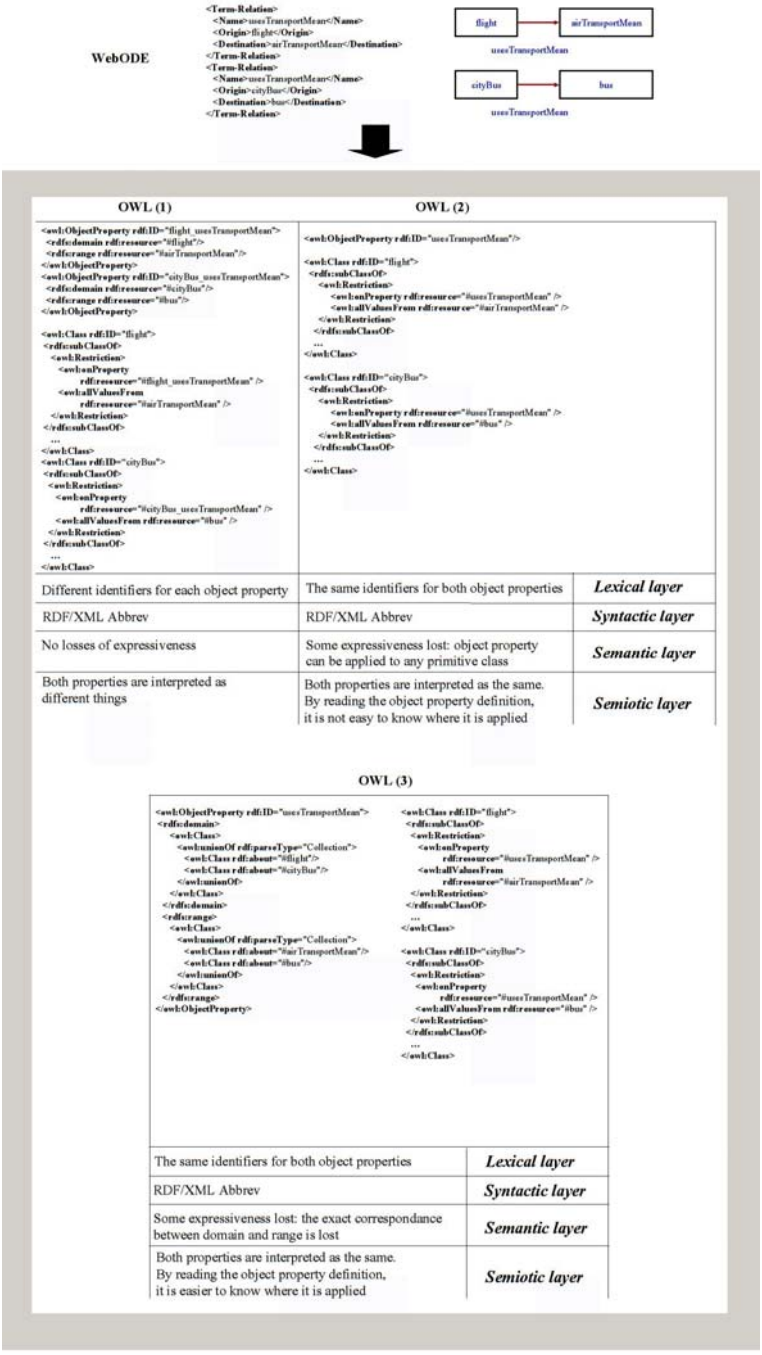

Figure 2: Example of translation decisions to be taken at several layers.

3 Description of ODEDialect

Taking into account the previous layers of transformations and the main characteristics of each of them, and the fact that implementing ontology translation decisions is usually a difficult task, we propose a set of languages that allow expressing such transformations declaratively. This set of languages will allow expressing transformations in the lexical layer (ODELex), in the syntax layer (ODESyntax), and in the semantic and pragmatic layers (ODESem). The last language deals with problems in both the semantic and pragmatic layers because both types of transformations are similar and hence require the same kind of language to be implemented.

In the following sections we will present the main features of each language, together with some examples extracted from our ontology translation system between WebODE and OWL DL.

3.1 ODELex: declarative specification of transformations at the lexical layer

Problems in the lexical layer are normally easy to handle, because it is usually enough to take into account the rules and conventions for creating identifiers and texts in the source and target formats.

The language ODELex allows specifying all the transformations to be made at this layer. This language is similar to lex [Lesk, 1975], a widely-used lexical analyser for building compilers. While lex is aimed at building compilers, ODELex is optimised for building ontology translation systems: it restricts some of the primitives available in lex and provides specific ones related to the construction of this type of systems. We will now describe the main parts of an ODELex specification.

An ODELex specification is composed of three parts, as shown in the following derivation rule[3], which contain: user code, declarations, and lexical rules. These parts are separated by the %% symbol. Besides, Java-style comments can be added at the beginning of the document, or inside the other parts, as will be seen later, in derivation rules 24, 25, and 26.

(1) *ODELexDocument* :: {*comment*} **%%** [*userCode*] **%%** [*declarations*] **%%** [*lexRules*]

User code. The first part of an ODELex document contains the user code, where users must include any Java functions used in the rest of the specification plus any Java import statements needed for these functions. These functions usually implement complex transformations to be made to ontology component identifiers or pieces of text, or they are used as a kind of macro definitions for a sequence of transformations that have to be performed using standard functions from the Java API.

(2) *userCode* :: {*comment*} **%(** *javaCode***)%**

[3] The following notation will be used to describe the derivation rules of the ODELex, ODESyntax and ODESem grammars: words in *italics* will be used for non-terminal symbols, words in **bold** font will be used for terminal symbols, alternatives will be represented with the | symbol, optional elements will be enclosed in square brackets **[ ]**, iterations of 0 or more items will be enclosed in braces **{ }**, and ranges of values will be enclosed in parenthesis and separated by hyphens **(-)**.

The derivation that corresponds to the symbol javaCode is not included here, since it corresponds to the grammar used for any Java file, and hence can be found in other documents that deal with that programming language. In our example, we have defined two Java functions:

- *convertToURI*, which transforms a string value into a corresponding string value that is a valid URI.
- *addNumber*, which adds the characters "_1" to the identifier that it receives as an input.

```
%(
import java.net.*;

private String convertToURI (String id){
    URI id_URI = new URI(URLEncoder.encode(id,"ISO-8859-1"))
    return id_URI.toString();
}

private String addNumber (String identifier){
    return (identifier + "_1");
}
)%
```

The definitions included in the user code part will be copied verbatim into the source file of the *LexicalMapper.java* file generated from this specification. In this part there is also the possibility of referencing transformation functions from a lexer created with typical lexical analysis tools such as lex [Lesk, 1975; Levine et al., 1992], JLex[4] [Berk, 1997], etc.

Declarations. This part of an ODELex specification contains the declaration of which ontology components (both from the source format and from the target format) will be dealt with by the lexical transformation tool, and of which ontology components of the target format cannot overlap (which means that they cannot share the same identifiers because they share the same scope).

(3) *declarations* :: {*comment* | *componentDecl* | *overlapDecl*}
(4) *componentDecl* :: *idComponent* [**%transient**] [*scopeDecl*]
(5) *scopeDecl* :: **%scope (** *idScope* {**,** *idScope*} **)**
(6) *idScope* :: *idComponent* | *id*
(7) *overlapDecl* :: **no-overlap (** *idComponent* **,** *idComponent* {**,** *idComponent*} **)**

In the example below, the declaration part defines seven WebODE ontology components (concepts, instance attributes, class attributes, ad hoc relations, instances, references, and axioms) and two types of values (values and documentation). With respect to OWL, it defines four ontology components (classes, object properties, datatype properties, instances) and two types of values (datatype values, and label and comments). It also states that the sets of identifiers of the four ontology components must be disjoint (they cannot overlap).

Three of the WebODE components are not first class citizens of the ontology, since they are defined inside the scope of others: instance and class attributes are

[4] http://www.cs.princeton.edu/~appel/modern/java/JLex/

defined inside the scope of concepts, and ad hoc relations are defined inside the scope of two concepts (their domain and range).

Finally, the *%transient* keyword states that once that we have performed a transformation of the corresponding component, it is not interesting to store the result of the transformation. In the example, this happens with attribute values and pieces of text used to document ontology components in both formats.

```
/* WebODE components */
WebODE.Concept
WebODE.InstanceAttribute %scope (WebODE.Concept)
WebODE.ClassAttribute %scope (WebODE.Concept)
WebODE.AdHocRelation %scope (WebODE.Concept,WebODE.Concept)
WebODE.Instance
WebODE.Reference
WebODE.Axiom
WebODE.Value %transient
WebODE.Documentation %transient

/* OWL components */
OWL.Class
OWL.ObjectProperty
OWL.DatatypeProperty
OWL.Instance
OWL.DatatypeValue %transient
OWL.LabelComment %transient
no-overlap   (OWL.Class, OWL.ObjectProperty,
                 OWL.DatatypeProperty, OWL.Instance)
```

If a component is not included in this declaration part, its identifier or the corresponding piece of text will not be transformed because it is legal in the target format, and hence do not have to be transformed. This part of the code will be used to generate the file *LexicalTypes.java* that contains an interface with the components to be used in the lexical transformations. This interface will be used by the rest of lexical tools, and also by the tools from other layers.

Lexical transformation rules. This part of an ODELex specification contains the actual transformations that have to be performed to each ontology component of the source format in order to obtain its correspondence in the target format.

(8) *lexRules* :: {*lexRule*}

(9) *lexRule* :: *ruleHeader* CR *init* CR *table* CR *repeated* CR *overlap* CR

(10) *ruleHeader* :: **%** *idComponent* **IDENTIFIER** {*idComponent* **IDENTIFIER**}

As shown in rule 9, for each component specified in the lexical rule header the following information must be specified:

- INIT: it specifies the initial transformation to be performed. For instance, in the example below we propose to transform the identifier of a WebODE ad hoc relation to an OWL ObjectProperty by converting the identifier to a URI, with the function *convertToURI* specified above.

  (11) *init* :: **INIT:{** *javaCode* **}**

- TABLE: if the component is not transient, it specifies the information to be stored so as to obtain it later either in the source format or in the target format. In

the example below we propose to store the ad hoc relation identifier, and its two associated concept identifiers in the table *WebODE.AdHocRelation*, and the corresponding identifier obtained from the transformation in the table *OWL.ObjectProperty*, maintaining the corresponding links to each other.

(12) *table* :: **TABLE:{** *tableDecl* **}**
(13) *tableDecl* :: **(** *tableColumn* **,** *tableColumn* **)**
(14) *tableColumn* :: **[** *idComponent* **,** *numberPosition* {**,** *numberPosition*}**]**
(15) *numberPosition* :: **%** *number* | **$** *number*

- REPEATED: if the component is not transient and cannot be repeated under certain circumstances, it specifies the alternative transformation to be made. For instance, if after the transformation the OWL object property identifier already existed as an OWL object property identifier and the WebODE ad hoc relation with the same ad hoc relation identifier and domain concept already existed, then we propose to maintain the identifier to be provided for the transformed object property (and we obtain it with the predefined function *GET*). The same applies if the same ad hoc relation identifier and range concept already existed. In another situation, a new identifier is created by adding the character "1" to the current transformed identifier.

(16) *repeated* :: **REPEATED:{** *transformation* {**,** *transformation*} **}**
(17) *transformation* :: *tablePatternColumn* **==> {** *javaCode* **}** |
default ==> { *javaCode* **}**
(18) *tablePatternColumn* :: **[** *idComponent* **,** *numberPatternPosition* {**,** *numberPatternPosition*}**]**
(19) *numberPatternPosition* :: **%** *number* | **_**

- OVERLAP: if the component is not transient and there cannot be overlaps in the target format, it specifies the transformation to be performed to the already generated identifier. This is repeated until there is no overlap. In the example, we propose to add a number to the OWL object property identifier until there is no collision with other class, datatype property and instance identifiers (not object properties).

(20) *overlap* :: **OVERLAP:{** *javaCode* **}**

```
%WebODE.AdHocRelation IDENTIFIER
     WebODE.Concept IDENTIFIER WebODE.Concept IDENTIFIER
  /* The 2nd and 3rd identifiers are of the domain and range
  concepts */
  INIT: {$1=convertToURI(%1)}
  TABLE:{([WebODE.AdHocRelation,%1,%2,%3],
          [OWL.ObjectProperty,$1])}
  REPEATED:
    {[WebODE.AdHocRelation,%1,%2,_] ==>

    {$1=GET([WebODE.AdHocRelation,%1,%2,_])},
       [WebODE.AdHocRelation,%1,_,%3] ==>

    {$1=GET([WebODE.AdHocRelation,%1,_,%3])},
       default ==> {$1=addNumber($1)}}
  OVERLAP: {$1=addNumber($1)}
```

This part of the ODELex specification will be used to generate most of the *LexicalMapper.java* file, which contains the lexical tools to be used by the tools generated by other layers in order to access the lexical information of each ontology component.

Finally, the following derivation rules are used for creating identifiers, Java-style comments, numbers and the end of line symbol. They have been used in the previous rules.

(21) *idComponent* :: *idFormat* **.** *id*
(22) *idFormat* :: (**A-Z**){**A-Z**}
(23) *id* :: (**A-Z**,**a-z**){**a-z**,**A-Z**,**0-9**}
(24) *comment* :: **/*** *textIncludingCR* ***/** | **//** *textWithoutCR*
(25) *textIncludingCR* :: {**a-z**,**A-Z**,**0-9**,**CR**}
(26) *textWithoutCR* :: {**a-z**,**A-Z**,**0-9**}
(27) *number* :: (**1-9**){**0-9**}
(28) *CR* :: **n**

Declarative specification of transformations at the syntactic layer

Given that one of our assumptions is that both the source and the target formats of the transformations have their own Java APIs defined, the transformations to be expressed in this layer are also simple, as occurred with the lexical layer. With regard to the source format, specifications at this layer describe the correspondence between each component and its accessors (either for the component itself, for all the components of a specific type, or for pieces of information of each component). With regard to the target format, specifications at this layer describe the correspondence between each component and its constructors, adding and updating methods, as well as the accessors that might be needed . Besides, in this layer it is also specified the correspondence between the attribute datatypes of the source and target formats.

As occurred in the lexical layer, in the syntactic layer we propose the use of the language ODESyntax, which allows specifying all the correspondences outlined above. This language is also based on another one available for building compilers, such as yacc [Johnson, 1975]. A specification in this language is also divided in several parts, which will be described in detail below, using examples that correspond to the WebODE export service to OWL DL. The following rule shows that there are five parts: user code, declarations, accessors, constructor and updates, and datatype transformations.

(1) *ODESyntaxDocument* :: {*comment*} **%%** [*userCode*] **%%** [*declarations*]
%% [*accessDecls*] **%%** [*updateDecls*] **%%** [*datatype*]

User code. Like with ODELex, the first part of an ODESyntax specification contains the user code, where all the auxiliary Java functions to be used in the rest of the specification are included. These functions usually implement complex syntactic transformations (e.g., of attribute datatypes) or they are used as a kind of macro definitions for transformations, accessors, updaters, etc., that have to be implemented as sequences of functions from the standard Java API or from the APIs of the source and target formats. The corresponding derivation rule is the same as in ODELex:

(2) *userCode* :: {*comment*} **%(** *javaCode* **)%**

The following examples, taken from the WebODE export service to OWL DL, show some auxiliary functions defined in this part:

- *getConcepts*, which receives as an input an ontology name and returns all the concepts in the WebODE ontology. This function is defined here because the current WebODE ontology access API does not provide such a method.
- *addComment*, which receives the class where the comment must be added and the text of the comment. This function is defined here because in the Jena API (used to generate the target OWL ontology) the language in which the comment text is available must be specified. This function ensures that this language is set to null. As we will see later, this could have been omitted, since this function is used mostly as a macro.
- *removeAllSuperclasses*, which receives an OWL class and removes all its references to its superclasses. This function is defined here because the Jena API does not provide a specific function for performing this action, and it will be used in one of the following parts of the ODESyntax specification.

```
%(
//Import statements for the source format
import es.upm.fi.dia.ontology.webode.service.*;
//Import statements for the target format
import com.hp.hpl.jena.ontology.*;

private Concept[] getConcepts(String ontology){
      return (Concept[] (ode.getTerms
                (ontology,
                 new int [] {TermTypes.CONCEPT})));
}

private void addComment(OntClass class, String comm) {
      class.addComment(comm,null);
        //the second argument specifies the language
}

private void removeAllSuperclasses(OntClass class){
      ExtendedIterator iter = class.listSuperClasses();
      if (iter!=null){
            while (iter.hasNext()){
                  class.removeSuperClass(
                          (OntClass)iter.next());
            }
      }
}
…
)%
```

The definitions included in the user code part will be copied verbatim into the source file of the *SyntaxMapper.java* file generated from this specification. We also consider the possibility of referencing transformation functions from a syntax analiser created with typical syntax analysis tools such as yacc [Johnson, 1975; Levine et al., 1992], JCup[5] [Hudson, 1999], etc.

[5] http://www.cs.princeton.edu/~appel/modern/java/CUP/

Declarations. Like in ODELex, this part of an ODESyntax specification contains the declaration of which ontology components will be dealt with in the specification. This component list does not need to contain the same components than the ODELex one, since it has a different purpose (syntactic transformations instead of lexical ones). Besides, it does not have to consider whether a component is transient or not, nor whether there can be overlap or not with other components, since these problems are already solved by the lexical layer specification. The following rules are used to create this declaration part of the ODESyntax document:

(3) *declarations* :: {*comment* | *namespaceDecl* | *componentDecl* }
(4) *namespaceDecl* :: **%NAMESPACE** *id javaPackage* ;
(5) *componentDecl* :: *idComponent* [*scopeDecl*] **:** *javaClassID* **CR**
(6) *scopeDecl* :: **%scope (** *idComponent* {**,** *idComponent*} **)**

The ODESyntax component list is usually based on the knowledge models of the source and target formats, and usually corresponds as well to their APIs. If there is a strong correspondence between the format API and its knowledge model (either of the source or of the target format), the transformations to be specified in ODESyntax are quite straightforward. If not, the transformations are more complex and hence more effort is needed to construct the ODESyntax specification.

The declaration part also includes a list of namespaces, which is used to abbreviate long Java packages in the rest of the specification. To refer to these namespaces, the corresponding namespace identifier must be placed between square brackets (e.g., *[ode]* to refer to *es.upm.fi.dia.ontology.webode.service.*).

The declaration part of the example below defines two namespaces (*ode* and *jenaOnt*), which correspond to the packages where the knowledge models of the source and target formats are defined. It also defines the ontology components from the source and target formats that will be used for the syntactic transformations, together with their scope, in case that they depend on another components, and with their corresponding Java class or interface. In the case that a component does not have an interface or class defined for it in the format API, we should create it in the user code part or in an external file, and make references to it from this specification.

```
%NAMESPACE ode es.upm.fi.dia.ontology.webode.service.;
%NAMESPACE jenaOnt com.hp.hpl.jena.ontology.;

/* WebODE components */
WebODE.Ontology         :[ode]OntologyDescriptor
WebODE.Concept    %scope (WebODE.Ontology)  :[ode]Concept
WebODE Group      %scope (WebODE.Ontology)  :[ode]Group
WebODE.InstanceAttribute
    %scope (WebODE.Concept,WebODE.Ontology)
                             :[ode]InstanceAttributeDescriptor
WebODE.ClassAttribute
    %scope (WebODE.Concept)    :[ode]ClassAttributeDescriptor
WebODE.AdHocRelation
    %scope (WebODE.Concept,WebODE.Concept)
                               :[ode]TermRelation
WebODE.Reference
    %scope (WebODE.Ontology)   :[ode]ReferenceDescriptor
```

```
WebODE.Axiom
    %scope (WebODE.Ontology)   :[ode]FormulaDescriptor
WebODE.Instance
    %scope (WebODE.Ontology,WebODE.InstanceSet)
                               :[ode]Instance
WebODE.InstanceSet
    %scope (WebODE.Ontology)   :[ode]InstanceSet
/* OWL components */
OWL.Ontology                   :[jenaOnt]Ontology
OWL.Class                      :[jenaOnt]OntClass
OWL.ObjectProperty             :[jenaOnt]ObjectProperty
OWL.DatatypeProperty           :[jenaOnt]DatatypeProperty
OWL.Instance                   :[jenaOnt]Individual
```

Unlike in ODELex, the declaration part of an ODESyntax specification must include all the components that will be managed by the syntax transformations.

Accessor methods. For each ontology component of the source and target formats that has been declared in the previous piece of code, this part of an ODESyntax specification may declare the methods to be used for the following purposes:

- To access all the ontology components of a specific type (e.g., a method that retrieves all the concepts of an ontology).
- To access a specific ontology component of a specific type (e.g., a method that retrieves a specific concept of an ontology, given its identifier).
- To access different pieces of information of the ontology component (e.g., methods or properties to access a concept identifier, a concept description, the superclasses of a concept, etc.).

The following rules show how we can declare these methods:

(7) accessDecls :: {accessDecl}
(8) accessDecl :: header **CR** [all **CR**] [individual **CR**] [information **CR**]
(9) *header* :: **%** *idComponent* **IDENTIFIER** {*idComponent* **IDENTIFIER**}
(10) *all* :: **ALL:{** *functionDecl* {**;** *functionDecl* } **}**
(11) individual :: **INDIVIDUAL:{** functionDecl {**;** functionDecl } **}**
(12) functionDecl :: number **:** id **(** parameters **) :** javaClassID array
(13) *parameters* :: **%** *number* { **,** **%** *number* } | λ
(14) *array* :: **[]** | λ
(15) information :: **INFORMATION:{** informDecl {**;** informDecl } **}**
(16) informDecl :: id **:** id [**(** parameters **)**] **:** javaClassID array

As shown in the previous rules, the first group of accessors will be included in the *ALL* keyword, the second group will be included in the *INDIVIDUAL* keyword, and the third group will be included in the *INFORMATION* keyword. For each method in the first and second groups, we will indicate a number that identifies the method (because there can be different methods with the same purpose), the method name and parameters, and the object that it returns. For the third group, we specify the piece of information's identifier (which determines how we will access this information from the semantic and pragmatic layers), the property or method to be used to access to that information given a specific object of that type, and the datatype.

The following example shows the declaration corresponding to an ontology component of the source format: a WebODE instance attribute. It shows that instance attributes are defined inside the scope of an ontology and of a concept. There are two

methods that allow accessing all the instance attributes of a concept in an ontology: the first one is used to access the instance attributes that are defined explicitly in that concept, and the second one returns also those inherited through the concept taxonomy. Both of them return an array of objects of the class *InstanceAttributeDescriptor*.

There is one method to get a specific instance attribute, given the ontology name, the concept name and the attribute name.

Finally, the following information can be accessed from an instance attribute: the concept to which it belongs, the attribute name, the attribute description, its type, its maximum and minimum cardinality, its maximum and minimum value, and the set of explicit values. They are accessed using properties of the Java class *InstanceAttributeDescriptor*, except for the attribute type, which is accessed with an ad hoc function defined in the user code part.

```
%WebODE.InstanceAttribute IDENTIFIER
 WebODE.Concept IDENTIFIER
 WebODE.Ontology IDENTIFIER
ALL:
{1:getInstanceAttributes(%3,%2):
          [ode]InstanceAttributeDescriptor[];
      //gets instance attributes defined locally to concept
 2:getInstanceAttributes(%3,%2,true):
          [ode]InstanceAttributeDescriptor[]
      //gets also inherited instance attributes
  }
INDIVIDUAL:
{1:getInstanceAttribute(%3,%2,%1):
          [ode]InstanceAttributeDescriptor}
INFORMATION:
 {concept        :termName                          :String;
  Name           :name                              :String;
  Description    :description                       :String;
  Type           :getValueTypeName(valueType)       :String;
  MaxCard        :maxCardinality                    :int;
  MinCard        :minCardinality                    :int;
  MaxValue       :maxValue                          :float;
  MinValue       :minValue                          :float;
  Values         :values                            :String[]}
```

The following example shows the declaration that corresponds to an OWL class (an ontology component of the target format). It shows that OWL classes are not in the scope of other components, that there is a method that lists all the classes of an ontology, that there is also a method to access a specific class, given its identifier, and that we can access to the class name, description and to its superclasses. In some of these cases a *java.util.Iterator* is returned.

```
%OWL.Class IDENTIFIER
 ALL:            {1:listClasses()    :java.util.Iterator}
 INDIVIDUAL:     {1:getOntClass(%1)  :[jenaOnt]OntClass}
 INFORMATION:
     {name         :getURI()           :String;
      description  :listComments(null) :java.util.Iterator;
      subclassOf :listSuperClasses() :java.util.Iterator}
```

As commented for the declaration part, if a specific method or property that we want to specify in this part is not defined in the ontology access API of a format, we can define them either in the user code part of the ODESyntax specification or in a separate file that will extend the current API.

Constructor and update methods. For each ontology component of the target format that has been declared in the declaration part, this part of an ODESyntax specification may declare the methods that will be used for the following purposes:

- To create the ontology component in the target ontology (e.g., a constructor or a method that creates an OWL class). We use the keyword *CREATE*.
- To remove the ontology component from the target ontology (e.g., a method that removes a class from an OWL ontology). We use the keyword *REMOVE*.
- To add information about the ontology component (e.g., the method or property to be used to add information about the class documentation, the class superclasses, etc.). We use the keyword *ADD*.
- To remove all the information about a specific piece of information of the ontology component (e.g., the method or property to be used to remove all the class documentations, all the class superclasses, etc.). We use the keyword *REMOVEALL*.
- To remove a value from a specific piece of information of the ontology component (e.g., the method or property to be used to remove a specific class documentation, a specific class superclass, etc.). We use the keyword *REMOVEINDIVIDUAL*.

The following derivation rules show the syntax to be used to declare these constructors and update methods:

(17) *updateDecls* :: {*updateDecl*}
(18) *updateDecl* :: *header* **CR** [*create* **CR**] [*remove* **CR**] [*add* **CR**] [*removeall* **CR**] [*removeindividual* **CR**]
(19) create :: **CREATE:{** createremDecl {**;** createremDecl } **}**
(20) remove :: **REMOVE:{** createremDecl {**;** createremDecl } **}**
(21) createremDecl :: number **:** id **(** parameters **)**
(22) add :: **ADD:{** addremDecl {**;** addremDecl } **}**
(23) *removeall* :: **REMOVEALL:{** *addremDecl* {**;** *addremDecl* } **}**
(24) *removeindividual* :: **REMOVEINDIVID:{** *addremDecl* {**;** *addremDecl* } **}**
(25) *addremDecl* :: *id* **:** *id* **(** *parameters* **)**

For each method in the first and second groups, we will indicate a number that identifies the method (because there can be different methods with the same purpose), the method name and parameters. For the rest of groups, we specify the piece of information's identifier (which determines how we will access this information from the semantic and pragmatic layers), and the property or method to be used to add, remove all, and remove one of the value(s) of that piece of information, respectively. Not all the pieces of information must be specified in all the cases, but only those ones that will be used by the transformations in the semantic and pragmatic layers.

The following example shows the declaration corresponding to an OWL class. It shows that there is one method that allows creating classes in the target ontology, *createClass*, which receives as an input the identifier of the class. It also defines the methods that can be used to add a natural language description to the class and to

remove all the descriptions, and the ones to be used to add a superclass, to remove all the class superclasses, and to remove a specific superclass. The %1 parameter refers to the OWL class identifier, and the $1 in the parameter means that the value for this parameter (the actual description and the class identifier) will be provided by the semantic or pragmatic layers in the first order. We will see how to define this information in the semantic and pragmatic transformation layers.

```
%OWL.Class IDENTIFIER
CREATE:         {1:createClass(%1)}
ADD:            {description :addComment(%1,$1);
                 subclassOf  :addSuperClass(createClass($1))}
REMOVEALL:      {description :removeAllComments(%1);
                 subclassOf  :removeAllSuperclasses(%1)}
REMOVEINDIVIDUAL:{
          subclassOf:removeSuperClass(createClass($1))}
```

As commented for other parts of the specification, if a specific method or property used in this part is not defined in the ontology access API of a format, we can define them either in the user code part of the ODESyntax specification or in a separate file that will extend the current API.

Datatype transformations. This part of an ODESyntax specification contains the transformations that have to be made to the attribute datatypes that appear in the source format so as to transform them to the target format. This specific feature, which may have been included in the previous part of the specification (constructors and updater methods), aims to make it easier to specify these common transformations.

The following derivation rules are defined in the ODESyntax grammar to deal with this type of transformations:

(26) datatype :: {datatypeTransf}

(27) datatypeTransf :: " datatypeID " **:** " datatypeID " **;** |
default %1 : (*datatypeID* | **%1** | **{** *javaCode* **}**)

The following example shows how to specify these transformations. In the first column, we specify the String term used to identify the type in the source format. The second column specifies the correspondence for each datatype of the source format in the target format (in this example, all of them are XML Schema datatypes, since OWL DL only allows them to specify datatypes). The transformations will be checked sequentially, according to the order specified in this table. Besides, the "default" keyword can be used at the last line to specify any other kind of datatype that could be found.

```
/* Datatype transformations */
"boolean":  "http://www.w3.org/2001/XMLSchema#boolean";
"cardinal":
    "http://www.w3.org/2001/XMLSchema#nonNegativeInteger";
"integer":  "http://www.w3.org/2001/XMLSchema#integer";
"float":    "http://www.w3.org/2001/XMLSchema#float";
"string":   "http://www.w3.org/2001/XMLSchema#string";
"date":     "http://www.w3.org/2001/XMLSchema#date";
"range":    "http://www.w3.org/2001/XMLSchema#float";
"URL":      "http://www.w3.org/2001/XMLSchema#anyURI";
default %1: %1;
```

Finally, there are some non-terminal symbols that have been used by the previous rules of the ODESyntax grammar and whose derivation rules have not appeared yet. Besides the rules of this type that were defined for ODELex, we have the following additional ones:

(36) *javaClassID* :: **[** *id* **]** *id*
(37) *javaPackage* :: *id* **.** { *id* **.**}
(38) *datatypeID* :: {**a-z**,**A-Z**,**0-9**,**:**,/,**#**}

Declarative specification of transformations at the semantic and pragmatic layers

The previous sections have described how to specify declaratively the transformations to be made in the lexical and syntactic layers. The main objective of both transformation layers is to abstract the low-level details of the source and target formats (their syntax specific features, the restrictions and naming conventions of ontology component identifiers, etc.), so as to allow specifying the semantic and pragmatic transformations at a higher abstraction level.

In our proposal, we have considered the assumption that the problems to be solved in the semantic and pragmatic layers are the most important (and complex) ones, and they decide which transformations have to be performed by the ontology translation system. Hence the importance of abstracting low-level details of the source and target formats so as to allow knowledge engineers to focus on the transformations between their knowledge models.

The problems to be solved in the semantic layer are mainly related to complex transformations of expressions that go beyond the rather simple limits of syntax and whose general aim is usually the meaning preservation of the knowledge transformed. Some examples of such transformations are:

- Transform a component of the source format into several components in the target format. For instance, an OWL value restriction is transformed to a WebODE instance attribute and an explicit value assignment to that instance attribute.
- Transform a set of components of the source format into one component in the target format. For instance, a set of OWL *disjointWith* expressions and *subclass of* relationships are transformed into a WebODE disjoint decomposition.
- Transform a component of the source format to different components of the target format depending on some conditions. For instance, an RDF property must be transformed to a WebODE ad hoc relation if its range is an RDF class, and to a WebODE instance attribute if its range is a literal or an XML Schema datatype.

The problems to be solved in the pragmatic layer are those that permit both human users and ontology-based applications to notice as less differences as possible in the ontologies in the original and target formats. Examples of such transformations are:

- *Pre-processing transformations.* They usually consist in the creation of predefined ontology components in the target format so as to facilitate the transformation of other ontology components during the semantic processing. For example, creating the metaclass *:WebODEConcept* in Protégé so as to store

additional information about the ontology concepts being transformed from WebODE to Protégé.

- *Post-processing transformations*. They usually consist in transforming sets of expressions in the target format, which have been created as a result of the semantic transformations, into more legible (and usually equivalent) syntactic constructs in the target format. Some examples of such transformations are: transforming a set of WebODE formal axioms that define that several concepts are disjoint to each other into a WebODE concept group; transforming a domain of an OWL object property of the form `C1 U C2 U … U Cn` into a more general concept that subsumes the union of that concept; etc.
- *In-processing transformations*. Pragmatic decisions can be also taken during the processing of individual ontology components of the source format. These transformations differ from the semantic ones in the fact that they do not change the semantics of the knowledge transformed, but only how it can be interpreted by humans. Examples of such decisions are: whether to transform an RDF property without range to a WebODE instance attribute with type *String*, to a WebODE ad hoc relation with the ontology root concept as the range, or to both; which name should be assigned to the WebODE formal axiom derived from an OWL complete class definition; which name should be assigned to the WebODE anonymous class obtained from an OWL union of classes; etc.
- *Other transformations*. In this group we include any other transformations and decisions to be taken during the ontology translation process for pragmatic reasons, and that cannot be easily classified in the previous groups. For instance, hiding in the Protégé user interface the additional ontology components used to transform a WebODE ontology, so that users cannot find differences with the standard Protégé knowledge model.

The translation decisions in both layers will be specified with the same declarative language: ODESem. As with the other two languages presented in this chapter (ODELex and ODESyntax), we will now describe the parts in which this language is divided. The first derivation rule of the grammar shows that an ODESem specification is divided into three parts: user defined code, declarations and semantic and pragmatic rules:

(1) *ODESemDocument* :: {*comment*} **%%** [*userCode*] **%%** [*declarations*] **%%** [*semRules*]

User code. Like with the other two languages, the first part of an ODESem specification contains the user code, where all the auxiliary Java functions to be used in the rest of the specification are included. As we have commented above, there might be a need for performing complex transformations that might not be easily represented with the primitives used in the other parts of the specification. In this case, this user defined functions can be used from those parts. The grammar rule used for the user code part of an ODESem specification is like the ones used in the other two languages:

(2) *userCode* :: {*comment*} **%(** *javaCode* **)%**

The following examples, taken from the WebODE export service to OWL DL, show some auxiliary functions defined in this part:

- *obtainDatatypes* and *obtainDomains*, which receive as an input the identifier of an OWL datatype property and return all the datatypes and domains associated

to it, as stored in the lexical transformation mapping tools created from the ODELex specification. These functions are provided as shortcuts for the use of the predefined lexical function *obtainDistinctSourceParametersFromTargetId*.

- *obtainDatatype* and *obtainDomain*, which are similar to the previous ones, but return only one datatype and domain, respectively. These functions are used when the datatype property has only one datatype or range.
- *allDatatypesEqual* and *allDomainsEqual*, which receive as an input the identifier of an OWL datatype property and return whether it has only one datatype or not, and one domain or not, respectively. These functions are also shortcuts for the use of the predefined lexical function *obtainDistinctSourceParametersFromTargetId* and the check of the return value length.

```
private String[] obtainDatatypes (String propID){
      return
        obtainDistinctSourceParameterFromTargetID
      (LexicalMapper.OWL_DatatypeProperty,propID,2);
}
private String obtainDatatype (String propID){
      return
        obtainDistinctSourceParameterFromTargetID
      (LexicalMapper.OWL_DatatypeProperty,propID,2)[0];
}
private String[] obtainDomains (String propID){
      return
        obtainDistinctSourceParameterFromTargetID
      (LexicalMapper.OWL_DatatypeProperty,propID,1);
}
private String[] obtainDomain (String propID){
      return
        obtainDistinctSourceParameterFromTargetID
      (LexicalMapper.OWL_DatatypeProperty,propID,1)[0];
}
private boolean allDatatypesEqual (String propID){
  return
   (obtainDistinctSourceParameterFromTargetID
     (LexicalMapper.OWL_DatatypeProperty,propID,2).
      length<=1);
}
private boolean allDomainsEqual (String propID){
  return
   (obtainDistinctSourceParameterFromTargetID
     (LexicalMapper.OWL_DatatypeProperty,propID,1).
      length<=1);
}
```

As with the other two languages, the user code part will be copied verbatim into the source file of the *SemMapper.java* file generated from this specification.

Semantic and pragmatic transformation rule declarations and processing order. This part of an ODESem specification contains the declaration of the transformation rules that will be defined later, together with the order in which they have to be processed. Unlike in the rest of declarative specifications, where the order of the

definitions is not relevant, in this case the processing order of the semantic and pragmatic transformations might be relevant and hence it is considered in this specification.

The following rules of the ODESem grammar define how these rules and their processing order are defined:

(3) declarations :: {comment | ruleDecl }
(4) ruleDecl :: number **: %** id **;**

The following piece of code shows these declarations for the WebODE export service to OWL DL. This export service does not need to execute pre-processing rules. The rules 1 to 7 are in charge of the semantic and pragmatic transformations of the components found in the source format (the WebODE ontology). As we can infer from the rule names, the ontology translation system will first add general information about the ontology (ontology container), the classes, the object properties, the disjoint and exhaustive decompositions, and the ontology instances. Finally, three pragmatic post-processing transformations will be performed, in order to remove redundant domains in OWL datatype property definitions, and redundant domains and ranges in OWL object property definitions.

```
/* Semantic and pragmatic transform. rule declaration,
   and processing sequence */
1: %AddOntologyContainer;
2: %AddClasses;
3: %AddObjectProperties;
4: %AddDisjointDecompositions;
5: %AddExhaustiveDecompositions;
6: %AddInstances;
7: %PostProcessing_RemoveDPRedundantDomains;
8: %PostProcessing_RemoveOPRedundantDomains;
9: %PostProcessing_RemoveOPRedundantRanges;
```

Semantic and pragmatic transformation rules. The last part of an ODESem specification contains the rules to be applied in order to transform the ontology components in the source format to the corresponding ontology components in the target format. This part of the specification must at least contain the rules declared previously, plus any other auxiliary rules that can be called from these ones.

A semantic and/or pragmatic transformation rule is referenced to with an identifier, which is preceded by the symbol '%'. This is what is called header in the ODESem grammar, as shown below:

(7) semRules :: {semRule | comment}
(8) semRule :: header **CR** lhs **--> {** rhs **}**
(9) *header* :: **%** *id*

Besides, as shown in the previous derivation grammar rule, a transformation rule is define with two parts: the left hand side (LHS) and the right hand side (RHS):

- The LHS (Left Hand Side) of the rule, aka antecedent, contains the information needed to trigger the rule, which can be either the source ontology component to be transformed or a target component to be modified. In both cases, the antecedent is defined with the ontology component type, as defined in the ODESyntax specification, and with an identifier that will be used to refer to the specific component in the RHS (Right Hand Side) of the rule. If no information is needed to trigger the rule, the keyword *NULL* must be used.

  (10) *lhs* :: *idComponent id* | **NULL**

- The RHS of the rule, aka consequent, contains the sequence of actions that have to be performed in order to obtain the corresponding ontology component(s) in the target format.

The following rule of the ODESem grammar defines the set of actions that can be performed in the RHS of the transformation rule:

(11) *rhs* :: { *create* | *add* | *remove* | *removeall*
| *exec* | *ifThen* | *forEach*
| *error* | *assign* | *functionCall*} **;**

These actions can be grouped in three types: actions that create new ontology components in the target format, and add or remove components or information; actions that specify the control flow of the translation system; and general actions used to throw error messages, assign values to variables or call other functions, either predefined or defined by the user.

Let us start with the first group of actions, whose corresponding piece of grammar is shown below:

(18) *create* :: **CREATE (** *idComponent* **,** *number* **,** *var* {**,** *var*}**)**
(19) *add* :: **ADD (** *id* **,** *id* **,** (*create* | *var* | *getComponents*))
(20) *remove* :: **REMOVE (** *id* **,** *id* **,** *var* **)**
(21) *removeall* :: **REMOVEALL (** *id* **,** *id* **)**

- CREATE. It creates (and returns) an ontology component in the target ontology. This action receives as input parameters the ontology component type to be created, the number of the specific constructor to be used, and the rest of parameters needed to create the ontology component. All the information needed (the ontology component type, the number of the specific constructor and the number and order of the additional input parameters) must follow the restrictions coded in the ODESyntax specification.
- ADD. It adds one or several values to a specific property of an ontology component of the target ontology. The ontology component, the property and the value(s) are specified as input parameters.
- REMOVE. It removes a value from a specific property of an ontology component of the target ontology. The ontology component, the property and the value to be removed are specified as input parameters.
- REMOVEALL. It removes all the values from a specific property of an ontology component of the target ontology. The ontology component and the property are specified as input parameters.

The actions specified in the consequent of a transformation rule are executed sequentially. Besides, the following control flow structures can be used:

(22) *exec* :: **EXEC (%** *id* **,** *var* **)**
(15) *ifThen* :: **if (** *condition* **)** [**{**] *rhs* [**}**] **else** [**{**] *rhs* [**}**]
(16) *condition* :: *javaComparison* | *functionCall*
(17) *forEach* :: **forEach** *id* **IN** *var* [**{**] *rhs* [**}**]

- EXEC. It starts the execution of a rule, with the set of parameters that match its antecedent.
- If *condition* {*actions*} else {*actions*}. It specifies the set of actions to be performed if the condition specified is evaluated as true (*then* body), and, optionally, the set of actions to be performed if the condition specified is evaluated as false (*else* body).

- ForEach *variable* IN *set_variable* {*actions*}. It specifies the set of actions to be performed for each ontology component inside the multiple-valued variable.

Finally, any variable assignments can be used, using the form *var = value*, and the predefined functions *GETCOMPONENT* and *GETALLCOMPONENTS* can be used to obtain a specific component or all the components of either the source or the target format (with the parameters specified in the ODESyntax specification). The *ERROR* function can be used in cases where the options that allow executing it are not allowed. These actions are considered in the following ODESem grammar rules:

(12) assign :: id = { create | functionCall | getComponents}
(13) functionCall :: id **(** parameters **)**
(14) *parameters* :: *id* { **,** *id* } | λ
(26) *getComponents* :: **GETCOMPONENT (** *idComponent* **,** *id* **)** | **GETALLCOMPONENTS (** *idComponent* **,** *id* **)**

Below we provide the code of some transformation rules of the WebODE export service to OWL DL, specifically those that transform WebODE concepts and their instance attributes in OWL classes and datatype properties.

The first rule (*AddClasses*) appeared in the declaration and processing order part of the specification. For each concept defined in WebODE, it creates an OWL class with the concept name, it adds a natural language description, in case that it exists and that the concept is not an imported term from another ontology, and it states that it is a subclass of the parent concepts, as specified in the source format. Finally, for each instance attribute of the concept, the rule *ADDInstanceAttributes* is triggered.

```
%AddClasses
WebODE.Concept concept -->
    {C = CREATE(OWL.Class,1,concept.name);
     if (concept.description != null && !concept.isImported)
           ADD(C,description,concept.description);
     ADD(C,subClassOf,concept.parentConcepts);
     forEach ia IN concept.instanceAttributes

    EXEC(%AddInstanceAttributes,WebODE.InstanceAttribute,ia);
    }
```

The second rule (*AddInstanceAttributes*) did not appear in the declaration and processing order part of the specification, hence it is considered as an auxiliary rule. It creates an OWL datatype property out from a WebODE instance attribute, and implements a decision tree that depends on whether there are several instance attributes in WebODE that produce the same OWL identifier or not, and in the first case, whether all those instance attributes have the same datatype associated or not. Depending on these conditions, the rules *AddInstanceAttributes_1* or *AddInstanceAttributes_2* will be triggered, or an error will be obtained.

```
%AddInstanceAttributes
WebODE.InstanceAttribute instAttr -->
    {P = CREATE(OWL.DatatypeProperty,1,propertyID);
     if (targetIDhasSeveralSources (WebODE.InstanceAttribute,

    instAttr.name,instAttr.concept,instAttr.type) {
      if (allDatatypesEqual (WebODE.InstanceAttribute,
           instAttr.name,instAttr.concept,instAttr.type))
             EXEC(%AddInstanceAttributes_1,P);
           else
```

```
            ERROR("Datatype property multiple datatypes");
    }else
           EXEC(%AddInstanceAttributes_2,P);
    }
```

Finally, the last two auxiliary rules are in charge of updating the information that corresponds to the datatype property created by the previous one. The first one (*AddInstanceAttributes_1*) is used when there are several instance attributes in WebODE that produce the same identifier, since they share the same datatype. This rule creates the union of all the concepts where those instance attributes are defined and set it as the domain of the datatype property. It also establishes the range of the property as the datatype of all those instance attributes. Finally, for each OWL class in the domain of the property it creates an OWL *AllValuesFrom* restriction, and the corresponding minimum and maximum cardinality restrictions, and attach them as *subclass of* restrictions of the class. The second rule (*AddInstanceAttributes_2*) is used when there is only one instance attribute that produces that datatype property identifier. In that case, the rule establishes the domain and range of the property, and adds OWL *AllValuesFrom* and maximum and minimum cardinality restrictions to the corresponding domain class.

```
%AddInstanceAttributes_1
OWL.DatatypeProperty P -->
    {domains = obtainDomains(P);
     ADD(P,domain,CREATE(OWL.UnionOf,1,domains));
     ADD(P,range,obtainDatatype(P.name));
     forEach d IN domains {
        D = GETCOMPONENT(OWL.Class,d);
        ADD(D,subClassOf,

     CREATE(OWL.AllValuesFrom,1,P,obtainDatatype(P.name)));
        if (instAttr.minCard!=0)
            ADD(D,subClassOf,

    CREATE(OWL.MinCardRestriction,1,P,instAttr.minCard));
        if (instAttr.maxCard!=-1)
            ADD(D,subClassOf,

    CREATE(OWL.MaxCardRestriction,1,P,instAttr.maxCard));
     }
    }
%AddInstanceAttributes_2
OWL.DatatypeProperty P -->
    {ADD(P,domain,obtainDomain(P.name));
     ADD(P,range,obtainDatatype(P.name));
     D = GETCOMPONENT(OWL.Class,d);
     ADD(D,subClassOf,
      CREATE(OWL.AllValuesFrom,1,P, obtainDatatype(P.name)));
     if (instAttr.minCard!=0)
     ADD(D,subClassOf,
      CREATE(OWL.MinCardRestriction,1,P,instAttr.minCard));
     if (instAttr.maxCard!=-1)
     ADD(D,subClassOf,
      CREATE(OWL.MaxCardRestriction,1,P,instAttr.maxCard));
    }
```

Finally, we will show a transformation rule that corresponds to the pragmatic post-processing of datatype properties so as to remove the redundant domains. This transformation rule consists in checking whether the domain of a datatype property is an OWL union of classes. In that case, if any of the classes in that union is already a subclass of any of the other concepts in the union, then the class can be removed, since it is already considered in the domain.

```
%PostProcessing_RemoveDPRedundantDomains
OWL.DatatypeProperty P -->
    {forEach U in P.domain {
           if (U.isUnionOf)
                  forEach D in U.classes
                         forEach E in U.classes
                                if (E.isSubclassOf(D))
                                   REMOVE(U,class,E)
    }
```

4 Conclusions and Future Work

In this paper we have described the ODEDialect language, which is composed of the following three languages: ODELex, ODESyntax, and ODESem. These languages allow expressing declaratively the translation decisions to be made at four different layers (lexical, syntax, semantic, and pragmatic), based on existing classifications of semantic interoperability and on the theory of signs.

One of our assumptions in the development of these languages is that it is easier to construct and maintain ontology translation systems if their translation decisions are divided in the previous four layers. The types of transformations to be made at each layer are different, and consequently the languages for expressing transformations at each layer are different as well, except for the semantic and pragmatic ones.

Besides, one of the design issues considered for the creation of these languages has been that they should allow expressing declaratively most of the transformations to be made at each layer. However, at the same time it should be flexible enough to allow representing any type of transformations that could be needed for any translation system. We have achieved these objectives by proposing a large set of primitives in these languages and by allowing the inclusion of user-defined code in the general purpose programming language Java. Even if this may provide too much freedom for users, since they can use any Java construct to build their translation systems, our experience has shown that these constructs are only used in very few cases, and that due to the fact that many languages share common characteristics at different layers, these pieces of code can be normally reused across translation systems.

The ODEDialect language has been successfully used for the specification of the translation decisions of the import and export services of the WebODE ontology engineering workbench to and from OWL DL, RDF(S), and Protégé-Frames, and for their automatic generation and deployment in the workbench. These services have

been evaluated in the context of the EON interoperability workshops[6], with overall good results.

5 Related Work

Though there are no other integrated methods for building ontology translation systems available, we can find some technology that allows creating them. Specifically, we can cite two tools: Transmorpher and OntoMorph:

- *Transmorpher*[7] [Euzenat and Tardif, 2001] is a tool that facilitates the definition and processing of complex transformations of XML documents. Among other domains, this tool has been used in the context of ontologies, using a set of XSLT documents that are able to transform from one DL language to another, expressed in DLML[8]. This tool is aimed at supporting the "family of ontology languages" approach for ontology translation, described in [Euzenat and Stuckenschmidt, 2003]. The main limitation of this approach is that it only deals with problems in the semantic layer, not focusing on other problems related to the lexical, syntax and pragmatic layers.
- *OntoMorph* [Chalupsky, 2000] is a tool that allows creating translators declaratively. Transformations between the source and the target formats are specified by means of pattern-based transformation rules, and are performed in two phases: syntactic rewriting and semantic rewriting. The last one needs the ontology or part of it translated into PowerLoom, so that this KR system can be used for certain kinds of reasoning, such as discovering whether a class is subclass of another, whether a relation can be applied to a concept or not, etc. Since this tool is based on PowerLoom (and consequently on Lisp), it cannot handle easily all the problems that may appear in the lexical and syntax layers.

ODEDialect improves the support given by these systems by allowing the specification of transformations at more levels, so that ontology translation systems are easier to create, understand and maintain. Besides, ODEDialect can be applied to a wider range of formats, not necessarily based on XML or Lisp.

Another theoretical work that is worth mentioning, since it was applied for the formalization and development of KIF/Ontolingua translators to and from several languages (including LOOM, CLASSIC and EXPRESS) is the work on reversible grammars described in [van Baalen and Fikes, 1994]. Given a translation system between two models developed using a grammar based on Horn clauses, the authors define the set of conditions under which such a translation system can be used to perform the reverse translation. While this was an important contribution for ontology translation, it had an important limitation with respect to its applicability for the development of translation systems between any two models, since it is not always possible to specify all translation decisions using a grammar exclusively based on Horn clauses. Let us consider pragmatic decisions implemented with ODEDialect to remove facts from the target knowledge base and replace them with a different

[6] http://km.aifb.uni-karlsruhe.de/ws/eon2007

[7] http://transmorpher.inrialpes.fr/

[8] *Description Logic Markup Language.* http://co4.inrialpes.fr/xml/dlml/

knowledge representation primitive. This type of transformation cannot be represented with Horn clauses as this approach requires. Even for semantic decisions (which are the ones that this approach is focused on), they cannot be always reduced to Horn clauses either. For instance, it is not possible to express exclusively with a finite number of Horn clauses that if property P of a set of Protégé-Frames classes {C1,...,Cn} has always the same range (class D), then we can create in OWL an ObjectProperty P whose domain is the union of the set of classes {C1,...,Cn} and whose range is the class D. Only in the case where all decisions can be ultimately transformed into Horn clauses, this approach could provide the formal bases for determining whether there is a possibility of using that grammar to derive a reverse translator between the languages.

References

[Arpírez et al., 2003] Arpírez, J.C., Corcho, O., Fernández-López, M., Gómez-Pérez, A.: (2003) *WebODE in a nutshell*. AI Magazine 24(3):37-48. Fall 2003

[Baader, 1996] Baader, F.: (1996) *A Formal Definition for the Expressive Power of Terminological Knowledge Representation Languages*. Journal of Logic and Computation 6(1):33–54

[Baader et al., 2003] Baader, F., McGuinness, D., Nardi, D., Patel-Schneider, P.: (2003) *The Description Logic Handbook: Theory, implementation and applications*. Cambridge University Press, Cambridge, United Kingdom

[Bechhofer et al., 2001] Bechhofer, S., Horrocks, I., Goble, C., Stevens, R.: (2001) *OilEd: a reasonable ontology editor for the Semantic Web*. In: Baader F, Brewka G, Eiter T (eds) Joint German/Austrian conference on Artificial Intelligence (KI'01). Vienna, Austria. (Lecture Notes in Artificial Intelligence LNAI 2174) Springer-Verlag, Berlin, Germany, pp 396–408

[Borgida, 1996] Borgida, A.: (1996) *On the relative expressiveness of description logics and predicate logics*. Artificial Intelligence 82(1-2):353–367

[Brickley and Guha, 2004] Brickley, D., Guha, R.V.: (2004) *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C Recommendation. *http://www.w3.org/TR/PR-rdf-schema*

[Chalupsky, 2000] Chalupsky, H.: (2000) *OntoMorph: a translation system for symbolic knowledge*. In: Cohn AG, Giunchiglia F, Selman B (eds) 7th International Conference on Knowledge Representation and Reasoning (KR'00). Breckenridge, Colorado. Morgan Kaufmann Publishers, San Francisco, California, pp 471–482

[Corcho and Gómez-Pérez, 2000] Corcho, O., Gómez-Pérez, A.: (2000) *A Roadmap to Ontology Specification Languages*. In: Dieng R, Corby O (eds) 12th International Conference in Knowledge Engineering and Knowledge Management (EKAW'00). Juan-Les-Pins, France. Springer-Verlag, Lecture Notes in Artificial Intelligence (LNAI) 1937, Berlin, Germany, pp 80–96

[Dean and Schreiber, 2004] Dean, M., Schreiber, G.: (2004) *OWL Web Ontology Language Reference.* W3C Recommendation. *http://www.w3.org/TR/owl-ref/*

[Domingue, 1998] Domingue, J.: (1998) *Tadzebao and WebOnto: Discussing, Browsing, and Editing Ontologies on the Web*. In: Gaines BR, Musen MA (eds) 11th International Workshop on Knowledge Acquisition, Modeling and Management (KAW'98). Banff, Canada, KM4:1–20

[Euzenat, 2001] Euzenat, J.: (2001) *Towards a principled approach to semantic interoperability*. In: Gómez-Pérez A, Grüninger M, Stuckenschmidt H, Uschold M (eds) IJCAI2001 Workshop on Ontologies and Information Sharing, Seattle, Washington

[Euzenat and Stuckenschmidt, 2003] Euzenat, J., Stuckenschmidt, H. (2003) *The `family of languages' approach to semantic interoperability*. In: Omelayenko B, Klein M (eds) Knowledge transformation for the semantic web, IOS press, Ámsterdam, The Netherlands, pp49-63

[Euzenat and Tardif, 2001] Euzenat, J., Tardif, L.: (2001) *XML transformation flow processing*. Markup languages: theory and practice 3(3):285–311

[Farquhar et al. 1997] Farquhar, A., Fikes, R., Rice, J. (1997) *The Ontolingua Server: A Tool for Collaborative Ontology Construction*. International Journal of Human Computer Studies. 46(6):707–727

[Fernández-López et al., 2000] Fernández-López, M., Gómez-Pérez, A., Rojas-Amaya, M.D.: (2000) *Ontologies' crossed life cycles*. In: Dieng R, Corby O (eds) 12th International Conference in Knowledge Engineering and Knowledge Management (EKAW'00). Juan-Les-Pins, France. (Lecture Notes in Artificial Intelligence LNAI 1937) Springer-Verlag, Berlin, Germany, pp 65–79

[Genesereth and Fikes, 1992] Genesereth, M.R., Fikes, R.E.: (1992) *Knowledge Interchange Format. Version 3.0. Reference Manual*. Technical Report Logic-92-1. Computer Science Department. Stanford University, California, available at *http://meta2.stanford.edu/kif/Hypertext/kif-manual.html*

[Gómez-Pérez et al., 2003] Gómez-Pérez, A., Fernández-López, M., Corcho, O.: (2003) *Ontological Engineering: with examples from the areas of knowledge management, e-commerce and the Semantic Web*, Springer-Verlag, New York.

[Gómez-Pérez et al., 1997] Gómez-Pérez, A., Juristo, N., Montes, C., Pazos, J.: (1997) *Ingeniería del Conocimiento*. Centro de Estudios Ramón Areces

[Gruber, 1992] Gruber, T.R.: (1992) *Ontolingua: A Mechanism to Support Portable Ontologies*. Technical report KSL-91-66, Knowledge Systems Laboratory, Stanford University, Stanford, California. *ftp://ftp.ksl.stanford.edu/pub/KSL_Reports/KSL-91-66.ps*

[Gruber, 1993] Gruber, T.R.: (1993) *A translation approach to portable ontology specification*. Knowledge Acquisition 5(2):199–220

[Johnson, 1975] Johnson, S.C.: (1975) *Yacc: Yet Another Compiler Compiler*, Computing Science Technical Report No. 32, Bell Laboratories, Murray Hill, New Jersey

[Kifer et al., 1995] Kifer, M., Lausen, G., Wu, J.: (1995) *Logical Foundations of Object-Oriented and Frame-ased Languages*. Journal of the ACM 42(4):741–843

[Klein, 2001] Klein, M.: (2001) *Combining and relating ontologies: an analysis of problems and solutions*. In: Gómez-Pérez A, Grüninger M, Stuckenschmidt H, Uschold M (eds) IJCAI2001 Workshop on Ontologies and Information Sharing, Seattle, Washington

[Knublauch, 2003] Knublauch, H.: (2003) *Editing Semantic Web Content with Protégé: the OWL Plugin*. 6th Protégé workshop. Manchester, United Kingdom

[Lassila and Swick, 1999] Lassila, O., Swick, R.: (1999) *Resource Description Framework (RDF) Model and Syntax Specification*. W3C Recommendation. *http://www.w3.org/TR/REC-rdf-syntax/*

[Lenat and Guha, 1990] Lenat, D.B., Guha, R.V.: (1990) *Building Large Knowledge-based Systems: Representation and Inference in the Cyc Project*. Addison-Wesley, Boston, Massachusetts

[Lesk, 1975] Lesk, M.E.: (1975) *Lex - A Lexical Analyzer Generator*, Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, New Jersey

[MacGregor, 2001] MacGregor, R.: (2001) *Inside the LOOM classifier*. SIGART Bulletin 2(3):88–92

[Maedche et al. 2003] Maedche, A., Motik, B., Stojanovic, L., Studer, R., Volz, R.: (2003) *Ontologies for Enterprise Knowledge Management*. IEEE Intelligent Systems 18(2):26–33

[Morris, 1938] Morris, C.W.: (1938) *Foundations of the theory of signs.* In: Neurath O, Carnap R, Morris CW (eds) International encyclopedia of unified science. Chicago University Press [reprinted in C W Morris 1971 *Writings on the theory of signs*. Mouton, The Hague]

[Motta, 1999] Motta, E.: (1999) *Reusable Components for Knowledge Modelling: Principles and Case Studies in Parametric Design*. IOS Press. Amsterdam, The Netherlands

[Noy et al. 2000] Noy, N.F., Fergerson, R.W., Musen, M.A.: (2000) *The knowledge model of Protege-2000: Combining interoperability and flexibility*. In: Dieng R, Corby O (eds) 12th International Conference in Knowledge Engineering and Knowledge Management (EKAW'00). Juan-Les-Pins, France. (Lecture Notes in Artificial Intelligence LNAI 1937) Springer-Verlag, Berlin, Germany, pp 17–32

[Schreiber et al., 1999] Schreiber, G., Akkermans, H., Anjewierden, A., de Hoog, R., Shadbolt, N., van de Velde, W., Wielinga, B.: (1999) *Knowledge engineering and management. The CommonKADS Methodology*. MIT press, Cambridge, Massachusetts

[Sure et al., 2002] Sure, Y., Staab, S., Angele, J.: (2002) *OntoEdit: Guiding Ontology Development by Methodology and Inferencing*. In: Meersman R, Tari Z (eds) Confederated International Conferences CoopIS, DOA and ODBASE 2002, University of California, Irvine. (Lecture Notes in Computer Science LNCS 2519) Springer-Verlag, Berlin, Germany, pp 1205–1222

[Swartout et al., 1997] Swartout, B., Ramesh, P., Knight, K., Russ, T.: (1997) *Toward Distributed Use of Large-Scale Ontologies*. In: Farquhar A, Gruninger M, Gómez-Pérez A, Uschold M, van der Vet P (eds) AAAI'97 Spring Symposium on Ontological Engineering. Stanford University, California, pp 138–148

[Tempich et al., 2005] Tempich, C., Pinto, H.S., Sure, Y., Staab, S.: (2005) *An Argumentation Ontology for DIstributed, Loosely-controlled and evolvInG Engineering processes of oNTologies (DILIGENT).* ESWC 2005: 241-256

[Valente et al., 1999] Valente, A., Russ, T., MacGregor, R.M., Swartout, W.R.: (1999) *Building and (Re)Using an Ontology of Air Campaign Planning.* IEEE Intelligent Systems. 1999

[Van Baalen and Fikes, 1994] Van Baalen, J., Fikes, R.E.: (1994) *The Role of Reversible Grammars in Translating Between Representation Languages.* In Doyle J, Sandewall E, Torasso P (editors): Proceedings of KR94: Principles of Knowledge Representation and Reasoning, pp 562-571